Representing Aliasing in GCC

Diego Novillo

dnovillo@redhat.com

Red Hat Canada

Gelato ICE 2007

San Jose, California, April 2007

Memory expressions in GIMPLE

- At most one memory load and one memory store per statement
  - Loads only allowed on RHS of assignments
  - Stores only allowed on LHS of assignments
- Gimplifier will enforce this property
- Dataflow on memory represented explictly
  - Factored Use-Def (FUD) chains or “Virtual SSA”
  - Requires a symbolic representation of memory

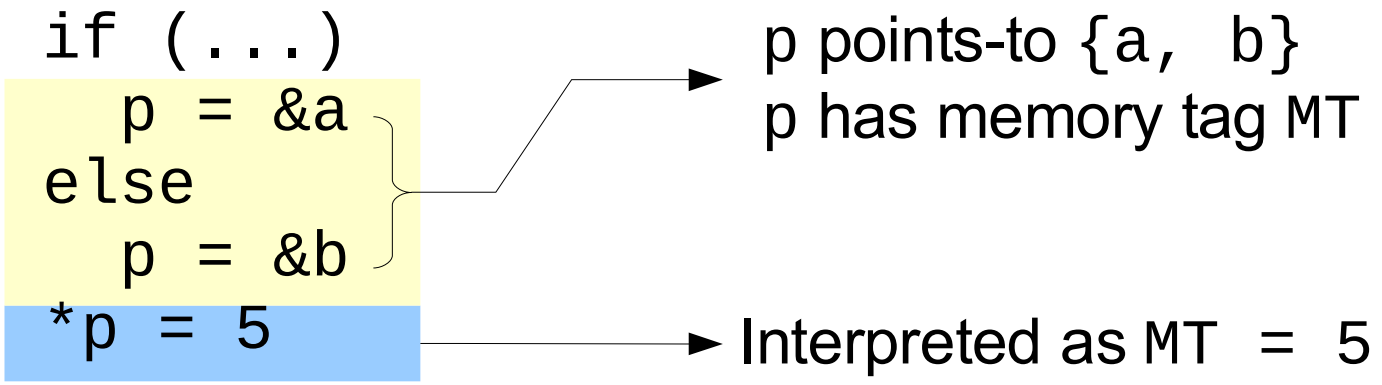
Symbolic Representation of Memory

- Aliased memory referenced via pointers
- GIMPLE only allows single-level pointers

Invalid	Valid
`**p`	`t.1 = *p` `*t.1`
`*(a[3].ptr)`	`t.1 = a[3].ptr` `*t.1`

Symbolic Representation of Memory

- Pointer P is associated with memory tag MT
  - MT represents the set of variables pointed-to by P
- So *P is a reference to MT



Associating Memory with Symbols

- Alias analysis
  - Builds points-to sets and memory tags
- Structural analysis
  - Builds field tags (sub-variables)
- Operand scanner
  - Scans memory expressions to extract tags
  - Prunes alias sets based on expression structure

Alias Analysis

- GIMPLE only has single level pointers.
- Pointer dereferences represented by artificial symbols ⇒ ***memory tags*** (MT).
- If p points-to x ⇒ p's tag is aliased with x.

```
# MT = VDEF <MT>
*p = ...
```

- Since MT is aliased with x:

```
# x = VDEF <x>
*p = ...
```

Alias Analysis

- Points-to alias analysis (PTAA)
  - Based on constraint graphs
  - Field and flow sensitive, context insensitive
  - Intra-procedural (inter-procedural in 4.2)
  - Fairly precise
- Type-based analysis (TBAA)
  - Based on input language rules
  - Field sensitive, flow insensitive
  - Very imprecise

Alias Analysis

- Two kinds of pointers are considered
  - Symbols: Points-to is flow-insensitive
    - Associated to Symbol Memory Tags (SMT)
  - SSA names: Points-to is flow-sensitive
    - Associated to Name Memory Tags (NMT)
- Given pointer dereference $*ptr_{42}$
  - If ptr_{42} has NMT, use it
  - If not, fall back to SMT associated with `ptr`

IL Representation

```
foo (i, a, b, *p)
{
  p =(i > 10) ? &a : &b
  *p = 3
  return a + b
}
```

```
foo (i, a, b, *p)
{
  p = (i > 10) ? &a : &b

  # a = VDEF <a>
  # b = VDEF <b>
  *p = 3

  # VUSE <a>
  t1 = a

  # VUSE <b>
  t2 = b

  t3 = t1 + t2
  return t3
}
```

Virtual SSA Form

- VDEF operand needed to maintain DEF-DEF links
- They also prevent code movement that would cross stores after loads
- When alias sets grow too big, static grouping heuristic reduces number of virtual operators in aliased references

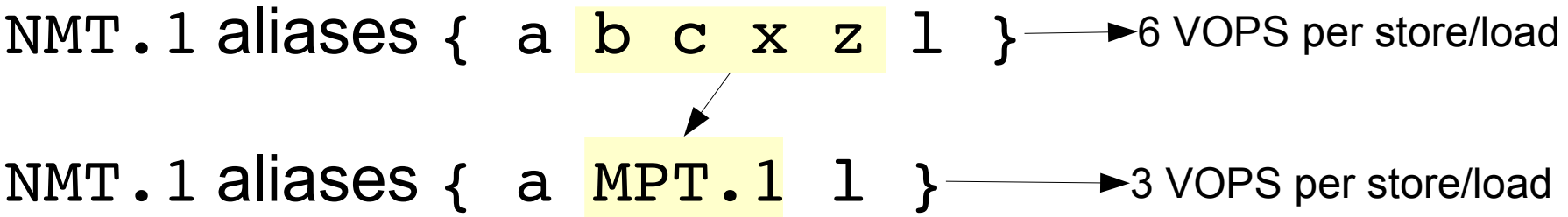
```
foo (i, a, b, *p)
{
  p_2 = (i_1 > 10) ? &a : &b

  # a_4 = VDEF <a_11>
  a = 9;

  # a_5 = VDEF <a_4>
  # b_7 = VDEF <b_6>
  *p = 3;

  # VUSE <a_5>
  t1_8 = a;

  t3_10 = t1_8 + 5;
  return t3_10;
}
```

Virtual SSA – Problems

- Big alias sets → Many virtual operators
  - Unnecessarily detailed tracking
  - Memory
  - Compile time
  - SSA name explosion
- Static alias grouping helps
  - Reverse role of alias tags and alias sets
  - Approach convoluted and too broad

Memory SSA

- Attempts to reduce the number of virtual operators in the presence of big alias sets
- Main idea
  - Alias sets are reduced by partitioning
  - Partitions affect representation **not** points-to results



Partitioning schemes

- Dynamic
  - Every store generates a different partition
  - Stores generate a single SSA name N
  - N becomes `currdef` for all the affected symbols
  - Loads are handled as usual
- Static
  - Partitions are determined before SSA renaming
  - Associations stay fixed

Dynamic partitioning

```
# .MEM_10 = VDEF <.MEM_0>
*p_3 = ...

# .MEM_11 = VDEF <.MEM_0>
*q_4 = ...

# b_12 = VDEF <.MEM_10>
b = ...

# .MEM_13 = VDEF <.MEM_10, b_12>
*p_3 = ...

# VUSE <.MEM_13>
t_14 = b

# VUSE <.MEM_11>
t_15 = o
```

p_3 points-to { a, b, c }

q_4 points-to { n, o, p }

At most one VDEF and one VUSE per statement

Virtual operators may refer to more than one operand

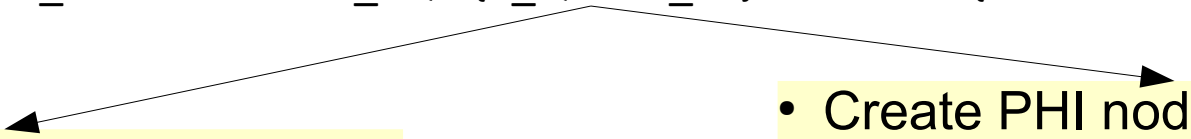
Factored stores create “sinks” that group multiple incoming names

Dynamic partitioning

- Advantages
  - Stores generate exactly one SSA name
  - Loads not reached by unrelated SSA names (no false conflicts)
- Disadvantages
  - Creates overlapping live ranges (OLR)
  - SSA renaming more complex
  - PHI nodes are a problem

Dynamic partitioning

```
if (…)
     # MEM_10 = VDEF <…>
   *p_3 = …→ STORES { a b c d }
else
     # a_2 = VDEF <…>
     a = …
     # MEM_11 = VDEF <…>
   *q_5 = …→ STORES { a d }
endif
MEM_13 = PHI <MEM_10, {a_2, MEM_11}> STORES { a b c d }
```



- Insert one PHI node per symbol
- Defeats the factoring
- Generates even more VOPS

- Create PHI nodes for MEM
- Creates PHI arguments with multiple reaching definitions
- Leads to splitting and fixup problems

Static partitioning

- Partitions are symbols, not SSA names
- Association is done *before* SSA renaming
- Advantages
  - SSA renaming not affected
  - No OLR for virtual SSA names
- Disadvantages
  - False conflicts due to partitioning

Static vs Dynamic partitioning

p_9 points to { a b }

Dynamic partitioning
Stores to a and b do not conflict

```
# MEM_3 = VDEF <...>
*p_9 = ...
# a_5 = VDEF <MEM_3>
a = ...
# b_6 = VDEF <MEM_3>
b = ...
# VUSE <a_5>
... = a
```

Static partition MPT { a b }
Stores to a and b **do** conflict

```
# MPT_3 = VDEF <...>
*p_9 = ...
# MPT_5 = VDEF <MPT_3>
a = ...
# MPT_6 = VDEF <MPT_5>
b = ...
# VUSE <MPT_6>
... = a
```

Heuristic for static partitioning

- Goal: minimize false conflicts introduced by partitions
  - Partition as few symbols as possible
  - Only partition uninteresting symbols
- Partitioning algorithm
  1. Gather statistics on loads and stores (direct loads/stores, indirect load/stores, execution frequency, etc)
  2. Sort list by increasing score (try not to partition symbols with high scores)
  3. Partition until number of loads/stores below threshold

Hybrid partitioning

- Work in progress
- Use static partitioning to avoid the problems with PHI nodes from dynamic partitions
- PHI arguments with multiple reaching defs

```
if (...)
  # x_4 = VDEF < ... >
  x = ...
  # y_5 = VDEF <...>
else
  # x_7 = VDEF <...>
endif
MPT_10 = PHI <{x_4, y_5}, x_7>
```